IT-314 Software Engineering

Lab - 08 Functional Testing (Black-Box)

Name : Shyam Ghetiya

Student-Id : 202201161

Aim: Functional Testing (Black-Box)

Q.1. **Consider a program for determining the previous date. Its input is triple of day, month and year with the following ranges 1 <= month <= 12, 1 <= day <= 31, 1900 <= year <= 2015.The possible output dates would be previous date or invalid date. Design the equivalence class test cases? Write a set of test cases (i.e., test suite) – specific set of data – to properly test the programs.**
Your test suite should include both correct and incorrect inputs. 1. Enlist which set of test cases have been identified using Equivalence Partitioning and Boundary Value Analysis separately. 2. Modify your programs such that it runs, and then execute your test suites on the program. While executing your input data in a program, check whether the identified expected outcome (mentioned by you) is correct or not.

❖ Equivalence class partitioning :

Equivalence Class	Description
E1	Month less than 1, Day less than 1, and Year less than 1900.
E2	Month less than 1, Day less than 1, and Year between 1900 and 2015 (inclusive).
E3	Month less than 1, Day less than 1, and Year greater than 2015.
E4	Month less than 1, Day between 1 and 31 (inclusive), and Year less than 1900.
E5	Month less than 1, Day between 1 and 31 (inclusive), and Year between 1900 and 2015 (inclusive).
E6	Month less than 1, Day between 1 and 31 (inclusive), and Year greater than 2015.
E7	Month less than 1, Day greater than 31, and Year less than 1900.
E8	Month less than 1, Day greater than 31, and Year between 1900 and 2015 (inclusive).
E9	Month less than 1, Day greater than 31, and Year greater than 2015.
E10	Month between 1 and 12 (inclusive), Day less than 1, and Year less than 1900
E11	Month between 1 and 12 (inclusive), Day less than 1, and Year between 1900 and 2015 (inclusive).
E12	Month between 1 and 12 (inclusive), Day less than 1, and Year greater than 2015.

E13	Month between 1 and 12 (inclusive), Day between 1 and 31 (inclusive), and Year less than 1900.
E14	Month between 1 and 12 (inclusive), Day between 1 and 31 (inclusive), and Year between 1900 and 2015 (inclusive).
E15	Month between 1 and 12 (inclusive), Day between 1 and 31 (inclusive), and Year greater than 2015.
E16	Month between 1 and 12 (inclusive), Day greater than 31, and Year less than 1900.
E17	Month between 1 and 12 (inclusive), Day greater than 31, and Year between 1900 and 2015 (inclusive).
E18	Month between 1 and 12 (inclusive), Day greater than 31, and Year greater than 2015.
E19	Month greater than 12, Day less than 1, and Year less than 1900.
E20	Month greater than 12, Day less than 1, and Year between 1900 and 2015 (inclusive).
E21	Month greater than 12, Day less than 1, and Year greater than 2015.
E22	Month greater than 12, Day between 1 and 31 (inclusive), and Year less than 1900.
E23	Month greater than 12, Day between 1 and 31 (inclusive), and Year between 1900 and 2015 (inclusive).
E24	Month greater than 12, Day between 1 and 31 (inclusive), and Year greater than 2015.
E25	Month greater than 12, Day greater than 31, and Year less than 1900.

❖ Equivalence test cases analysis:

Equivalence Class	Test Case	Month	Day	Year	Valid/Invalid	Description
E1	TC1	0	0	1800	Invalid	Month, day, and year are all out of valid range.
E1	TC2	1	1	1800	Valid	Month and day valid, year still less than 1900 but acceptable.
E2	TC3	0	0	1950	Invalid	Month and day are invalid, though the year is in the correct range.
E2	TC4	1	1	1950	Valid	Month and day valid, year within the specified range.
E3	TC5	0	0	2020	Invalid	Month and day invalid, year out of valid range.
E3	TC6	1	1	2020	Valid	Month and day valid, year out of range but acceptable.
E4	TC7	0	15	1850	Invalid	Month invalid, but day is within range. Year out of range.
E4	TC8	1	15	1850	Valid	Month and day valid, year still before 1900 but acceptable.
E5	TC9	0	15	2000	Invalid	Month invalid, day and year valid.
E5	TC10	1	15	2000	Valid	Month, day, and year are all valid.
E6	TC11	0	15	2021	Invalid	Month invalid, day and year valid.
E6	TC12	1	15	2021	Valid	Month, day, and year valid.
E7	TC13	0	32	1850	Invalid	Month and day invalid, year out of range.
E7	TC14	1	31	1850	Valid	Month and day valid, year still before 1900 but acceptable.
E8	TC15	0	32	1950	Invalid	Month and day invalid, year valid.

E8	TC16	1	31	1950	Valid	Month, day, and year are all valid.
E9	TC17	0	32	2021	Invalid	Month and day invalid, year valid.
E9	TC18	1	31	2021	Valid	Month, day, and year valid.
E10	TC19	1	0	1850	Invalid	Month valid, day and year out of range.
E10	TC20	1	1	1850	Valid	Month and day valid, year still before 1900 but acceptable.
E11	TC21	1	0	1950	Invalid	Month valid, day invalid, year valid.
E11	TC22	1	1	1950	Valid	Month, day, and year are all valid.
E12	TC23	1	0	2020	Invalid	Month valid, day invalid, year valid.
E12	TC24	1	1	2020	Valid	Month, day, and year are all valid.
E13	TC25	1	15	1850	Valid	Month, day, and year valid except year is before 1900.
E14	TC26	5	10	2010	Valid	Month, day, and year are all valid.
E15	TC27	5	10	2020	Valid	Month, day, and year valid except year is after 2015.
E16	TC28	1	32	1850	Invalid	Month valid, day invalid, year valid except year before 1900.
E17	TC29	1	32	2000	Invalid	Month valid, day invalid, year valid.
E18	TC30	1	32	2020	Invalid	Month valid, day invalid, year valid.
E19	TC31	13	0	1850	Invalid	Month and day invalid, year out of range.
E20	TC32	13	0	1950	Invalid	Month and day invalid, year valid.
E21	TC33	13	0	2020	Invalid	Month and day invalid, year valid.
E22	TC34	13	15	1850	Invalid	Month invalid, day valid, year out of range.
E23	TC35	13	15	2000	Invalid	Month invalid, day and year valid.

E24	TC36	13	15	2020	Invalid	Month invalid, day and year valid.
E25	TC37	13	32	1850	Invalid	Month invalid, day and year out of range.
E25	TC38	12	31	1850	Valid	Month and day valid, year out of range but acceptable.

❖ Boundary Test-Cases analysis:

Test Case	Month	Day	Year	Valid / Invalid	Description
TC1	0	1	1900	Invalid	Month is less than the minimum valid value.
TC2	1	1	1900	Valid	Minimum valid month, day, and year.
TC3	1	31	1900	Valid	Valid maximum day for January, minimum year.
TC4	1	32	1900	Invalid	Day exceeds maximum valid value for January.
TC5	1	1	2015	Valid	Minimum valid month, day, and maximum valid year.
TC6	1	31	2015	Valid	Valid maximum day for January, maximum year.
TC7	1	32	2015	Invalid	Day exceeds maximum valid value for January.
TC8	12	1	1900	Valid	Maximum valid month, minimum valid day and year.
TC9	12	31	1900	Valid	Maximum valid day for December, minimum year.
TC10	12	32	1900	Invalid	Day exceeds maximum valid value for December.
TC11	12	1	2015	Valid	Maximum valid month, minimum valid day, and maximum year.

TC12	12	31	2015	Valid	Valid maximum day for December, maximum year.
TC13	12	32	2015	Invalid	Day exceeds maximum valid value for December.
TC14	13	1	1900	Invalid	Month exceeds maximum valid value.
TC15	0	1	2015	Invalid	Month is less than the minimum valid value.
TC16	1	0	1900	Invalid	Day is less than the minimum valid value.
TC17	12	0	2015	Invalid	Day is less than the minimum valid value.
TC18	1	32	2015	Invalid	Day exceeds maximum valid value for January.
TC19	12	32	2015	Invalid	Day exceeds maximum valid value for December.
TC20	1	1	1899	Invalid	Year is below minimum valid value.
TC21	12	1	1899	Invalid	Year is below minimum valid value.
TC22	1	31	1899	Invalid	Year is below minimum valid value.
TC23	12	31	1899	Invalid	Year is below minimum valid value.
TC24	1	1	2016	Invalid	Year exceeds maximum valid value.
TC25	12	1	2016	Invalid	Year exceeds maximum valid value.

Q.2. Programs:

P1. The function linearSearch searches for a value v in an array of integers a. If v appears in the array a, then the function returns the first index i, such that a[i] == v; otherwise, -1 is returned.

```
int linearSearch(int v, int a[])
{
        int i = 0;
        while (i < a.length)
        {
                if (a[i] == v)
                return(i);
                i++;
        }
        return (-1);
}
```

Answer :

❖ Equivalence class partitioning :

1. **E1:** The array is empty (no elements).
2. **E2:** The value v exists in the array (multiple occurrences possible).
3. **E3:** The value v does not exist in the array (all elements are different).
4. **E4:** The array contains a single element equal to v.
5. **E5:** The array contains a single element not equal to v.

Equivalence Class	Test Case	Input Data (Array a, Value v)	Expected Outcome
The array is empty (no elements).	TC1	a = [], v = 20	-1
The value v exists in the array (multiple occurrences possible).	TC2	a = [4, 2, 20, 20, 5], v = 20	2
The value v does not exist in the array (all elements are different).	TC3	a = [3, 6, 8, 9], v = 20	-1
The array contains a single element equal to v.	TC4	a = [20], v = 20	0
The array contains a single element not equal to v.	TC5	a = [30], v = 20	-1

❖ Boundary Test-Cases analysis:

Boundary Condition	Test Case	Input Data (Array a, Value v)	Expected Outcome
Single element array, value present.	TC1	a = [20], v = 20	0
Single element array, value absent.	TC2	a = [20], v = 15	-1
Value is at the start of the array.	TC3	a = [1, 3, 5, 7], v = 1	0
Value is at the end of the array.	TC4	a = [2, 4, 6, 8], v = 8	3
Value is absent but close to elements in the array.	TC5	a = [1, 3, 5, 7], v = 4	-1

Linear-search Function :

```cpp
#include <iostream>
#include <vector>
using namespace std;

int linearSearch(const vector<int>& a, int v) {
   for (int i = 0; i < a.size(); ++i) {
       if (a[i] == v) {
           return i;  // Return the first index where a[i] == v
       }
   }
   return -1;  // If not found, return -1
}

int main() {
   // Test cases
   vector<int> arr1 = {};                    // Empty array
   vector<int> arr2 = {4, 2, 20, 20, 5};     // Value is present (multiple
occurrences)
   vector<int> arr3 = {3, 6, 8, 9};          // Value is not present
   vector<int> arr4 = {20};                   // Single element, value present
   vector<int> arr5 = {30};                   // Single element, value not present

   cout << linearSearch(arr1, 20) << endl;   // output -1
   cout << linearSearch(arr2, 20) << endl;   // output 2
   cout << linearSearch(arr3, 20) << endl;   // output -1
   cout << linearSearch(arr4, 20) << endl;   // output 0
   cout << linearSearch(arr5, 20) << endl;   // output -1

   return 0;
}
```

P2. The function countItem returns the number of times a value v appears in an array of integers a.

```
int countItem(int v, int a[])
{
	int count = 0;
	for (int i = 0; i < a.length; i++)
	{
		if (a[i] == v)
		count++;

	}
	return (count);

}
```

Answer :

❖ Equivalence class partitioning :

E1: The array is empty (no elements).
E2: The value v exists in the array multiple times (more than one occurrence).
E3: The value v exists in the array exactly once.
E4: The value v does not exist in the array (all elements are different).
E5: The array contains a single element that is equal to v.
E6: The array contains a single element that is not equal to v.

Equivalence Class	Test Case	Input Data (Array a, Value v)	Expected Outcome
The array is empty (no elements).	TC1	a = [], v = 5	0

The value v exists in the array multiple times (more than one occurrence).	TC2	a = [1, 2, 3, 5, 5, 5], v = 5	3
The value v exists in the array exactly once.	TC3	a = [1, 2, 3, 5], v = 3	1
The value v does not exist in the array (all elements are different).	TC4	a = [1, 2, 3, 4], v = 5	0
The array contains a single element that is equal to v.	TC5	a = [5], v = 5	1
The array contains a single element that is not equal to v.	TC6	a = [3], v = 5	0

❖ Boundary Test-Cases analysis:

Boundary Condition	Test Case	Input Data (Array a, Value v)	Expected Outcome
Single element array, value present.	TC1	a = [10], v = 10	1
Single element array, value absent.	TC2	a = [10], v = 5	0
Value present multiple times in a small array.	TC3	a = [2, 2, 2], v = 2	3
Value present once in a small array.	TC4	a = [1, 2, 3], v = 2	1
Value absent but close to elements in the array.	TC5	a = [1, 2, 3, 4, 5], v = 6	0

Count_Item function:

```cpp
#include <iostream>
#include <vector>
using namespace std;

int countItem(const vector<int>& a, int v) {
   int count = 0;
   for (int i = 0; i < a.size(); i++) {
       if (a[i] == v) {
           count++;  // Increment count for each occurrence of v
       }
   }
   return count;  // Return the total count
}

int main() {
   // Test cases
   vector<int> arr1 = {};                    // Empty array
   vector<int> arr2 = {1, 2, 3, 5, 5, 5};    // Value present multiple times
   vector<int> arr3 = {1, 2, 3, 5};          // Value present exactly once
   vector<int> arr4 = {1, 2, 3, 4};          // Value absent
   vector<int> arr5 = {5};                    // Single element, value present
   vector<int> arr6 = {3};                    // Single element, value absent

   cout << countItem(arr1, 5) << endl;   // output 0
   cout << countItem(arr2, 5) << endl;   // output 3
   cout << countItem(arr3, 3) << endl;   // output 1
   cout << countItem(arr4, 5) << endl;   // output 0
   cout << countItem(arr5, 5) << endl;   // output 1
   cout << countItem(arr6, 5) << endl;   // output 0

   return 0;
}
```

P3. The function binarySearch searches for a value v in an ordered array of integers a. If v appears in
the array a, then the function returns an index i, such that a[i] == v; otherwise, -1 is returned.
Assumption: the elements in the array are sorted in non-decreasing order.

```
int binarySearch(int v, int a[])
{
        int lo,mid,hi;
        lo = 0;
        hi = a.length-1;
        while (lo <= hi)
        {
                mid = (lo+hi)/2;
                if (v == a[mid])
                        return (mid);
                else if (v < a[mid])
                        hi = mid-1;
                else
                        lo = mid+1;

        }
        return(-1);
}
```

Answer :

❖ Equivalence class partitioning :

E1: The array is empty (no elements).
E2: The value v exists in the array (multiple occurrences).

E3: The value v exists in the array (exactly once).
E4: The value v does not exist in the array (less than the smallest element).
E5: The value v does not exist in the array (greater than the largest element).
E6: The value v is present at the start of the array.
E7: The value v is present at the end of the array.
E8: The value v is in the middle of the array.

Equivalence Class	Test Case	Input Data (Array a, Value v)	Expected Outcome
The array is empty (no elements).	TC1	a = [], v = 5	-1
The value v exists in the array (multiple occurrences).	TC2	a = [1, 2, 2, 2, 3], v = 2	1
The value v exists in the array (exactly once).	TC3	a = [1, 2, 3, 4, 5], v = 3	2
The value v does not exist in the array (less than the smallest element).	TC4	a = [1, 2, 3, 4, 5], v = 0	-1
The value v does not exist in the array (greater than the largest element).	TC5	a = [1, 2, 3, 4, 5], v = 6	-1
The value v is present at the start of the array.	TC6	a = [1, 2, 3, 4, 5], v = 1	0
The value v is present at the end of the array.	TC7	a = [1, 2, 3, 4, 5], v = 5	4
The value v is in the middle of the array.	TC8	a = [1, 2, 3, 4, 5], v = 3	2

❖ Boundary Test-Cases analysis:

Boundary Condition	Test Case	Input Data (Array a, Value v)	Expected Outcome
Single element array, value present.	TC1	a = [10], v = 10	0
Single element array, value absent.	TC2	a = [10], v = 5	-1
Value present at the start of the array.	TC3	a = [1, 2, 3, 4], v = 1	0
Value present at the end of the array.	TC4	a = [1, 2, 3, 4], v = 4	3
Value absent but close to the elements in the array.	TC5	a = [1, 2, 3, 4], v = 2.5	-1

Binary search Function :

```cpp
#include <iostream>
#include <vector>
using namespace std;

int binarySearch(const vector<int>& a, int v) {
   int lo = 0, hi = a.size() - 1;

   while (lo <= hi) {
       int mid = (lo + hi) / 2;
       if (v == a[mid]) {
           return mid;  // Value found, return index
       } else if (v < a[mid]) {
           hi = mid - 1;  // Search in the left half
       } else {
           lo = mid + 1;  // Search in the right half
       }
   }
   return -1;  // Value not found
}
```

```
int main() {
   // Test cases
   vector<int> arr1 = {};                    // Empty array
   vector<int> arr2 = {1, 2, 2, 2, 3};       // Value present multiple times
   vector<int> arr3 = {1, 2, 3, 4, 5};       // Value present exactly once
   vector<int> arr4 = {1, 2, 3, 4, 5};       // Value absent (less than smallest)
   vector<int> arr5 = {1, 2, 3, 4, 5};       // Value absent (greater than largest)
   vector<int> arr6 = {1, 2, 3, 4, 5};       // Value present at start
   vector<int> arr7 = {1, 2, 3, 4, 5};       // Value present at end
   vector<int> arr8 = {1, 2, 3, 4, 5};       // Value in the middle

   cout << binarySearch(arr1, 5) << endl;   // output -1
   cout << binarySearch(arr2, 2) << endl;   // output 1 (first occurrence)
   cout << binarySearch(arr3, 3) << endl;   // output 2
   cout << binarySearch(arr4, 0) << endl;   // output -1
   cout << binarySearch(arr5, 6) << endl;   // output -1
   cout << binarySearch(arr6, 1) << endl;   // output 0
   cout << binarySearch(arr7, 5) << endl;   // output 4
   cout << binarySearch(arr8, 3) << endl;   // output 2

   return 0;
}
```

P4. The following problem has been adapted from The Art of Software Testing, by G. Myers (1979). The function triangle takes three integer parameters that are interpreted as the lengths of the sides of a triangle. It returns whether the triangle is equilateral (three lengths equal), isosceles (two lengths equal), scalene (no lengths equal), or invalid (impossible lengths).

final int EQUILATERAL = 0;
final int ISOSCELES = 1;
final int SCALENE = 2;
final int INVALID = 3;

```
int triangle(int a, int b, int c)
{
        if (a >= b+c || b >= a+c || c >= a+b)
                return(INVALID);
        if (a == b && b == c)
                return(EQUILATERAL);
        if (a == b || a == c || b == c)
                return(ISOSCELES);

        return(SCALENE);
}
```

Answer :

❖ Equivalence class partitioning :

E1: All three sides are equal (equilateral triangle).
E2: Two sides are equal and one is different (isosceles triangle).
E3: All three sides are different (scalene triangle).
E4: The lengths cannot form a triangle (invalid case).
E5: One or more sides are zero or negative (invalid case).

Equivalence Class	Test Case	Input Data (Sides a, b, c)	Expected Outcome
All three sides are equal (equilateral triangle).	TC1	a = 5, b = 5, c = 5	"Equilateral"
Two sides are equal and one is different (isosceles triangle).	TC2	a = 5, b = 5, c = 3	"Isosceles"

All three sides are different (scalene triangle).	TC3	a = 5, b = 4, c = 3	"Scalene"
The lengths cannot form a triangle (invalid case).	TC4	a = 1, b = 2, c = 3	"Invalid"
One side is zero (invalid case).	TC5	a = 0, b = 5, c = 5	"Invalid"
One side is negative (invalid case).	TC6	a = -1, b = 2, c = 2	"Invalid"
Two sides are equal but do not form a triangle (invalid case).	TC7	a = 2, b = 2, c = 5	"Invalid"
The lengths can form a scalene triangle.	TC8	a = 7, b = 8, c = 9	"Scalene"

❖ Boundary Test-Cases analysis:

Boundary Condition	Test Case	Input Data (Sides a, b, c)	Expected Outcome
Minimum valid triangle (two sides equal, one side slightly longer).	TC1	a = 2, b = 2, c = 3	"Isosceles"
Minimum invalid triangle (two sides equal, one side equal to the sum).	TC2	a = 2, b = 2, c = 4	"Invalid"
Minimum valid scalene triangle (different lengths).	TC3	a = 2, b = 3, c = 4	"Scalene"
Minimum invalid case (one side is zero).	TC4	a = 0, b = 3, c = 4	"Invalid"
Minimum invalid case (negative side).	TC5	a = -1, b = 2, c = 2	"Invalid"

Triangle function :

```cpp
#include <iostream>
#include <string>
using namespace std;

const string EQUILATERAL = "Equilateral";
const string ISOSCELES = "Isosceles";
const string SCALENE = "Scalene";
const string INVALID = "Invalid";

string triangle(int a, int b, int c) {
   if (a <= 0 || b <= 0 || c <= 0 || a >= b + c || b >= a + c || c >= a + b) {
       return INVALID;  // Invalid case
   }
   if (a == b && b == c) {
       return EQUILATERAL;  // Equilateral case
   }
   if (a == b || a == c || b == c) {
       return ISOSCELES;    // Isosceles case
   }
   return SCALENE;        // Scalene case
}

int main() {
   // Test cases
   cout << triangle(5, 5, 5) << endl;   // output "Equilateral"
   cout << triangle(5, 5, 3) << endl;   // output "Isosceles"
   cout << triangle(5, 4, 3) << endl;   // output "Scalene"
   cout << triangle(1, 2, 3) << endl;   // output "Invalid"
   cout << triangle(0, 5, 5) << endl;   // output "Invalid"
   cout << triangle(-1, 2, 2) << endl;  // output "Invalid"
   cout << triangle(2, 2, 5) << endl;   // output "Invalid"
   cout << triangle(7, 8, 9) << endl;   // output "Scalene"

   return 0;
}
```

P5. The function prefix (String s1, String s2) returns whether or not the string s1 is a prefix of string s2 (you may assume that neither s1 nor s2 is null).

```
public static boolean prefix(String s1, String s2)
{
	if (s1.length() > s2.length())

	{
		return false;
	}
	for (int i = 0; i < s1.length(); i++)
	{
		if (s1.charAt(i) != s2.charAt(i))
		{
			return false;
		}
	}
	return true;
}
```

Answer :

❖ Equivalence class partitioning :

E1: s1 is an empty string (should return true for any s2).
E2: s1 is equal to s2 (should return true).
E3: s1 is a prefix of s2 (should return true).
E4: s1 is not a prefix of s2 (should return false).
E5: s1 is longer than s2 (should return false).

Equivalence Class	Test Case	Input Data (Strings s1, s2)	Expected Outcome
s1 is an empty string (true for any s2).	TC1	s1 = "", s2 = "Hello"	TRUE
s1 is equal to s2 (true).	TC2	s1 = "Hello", s2 = "Hello"	TRUE
s1 is a prefix of s2 (true).	TC3	s1 = "Hell", s2 = "Hello"	TRUE
s1 is not a prefix of s2 (false).	TC4	s1 = "Hello", s2 = "World"	FALSE
s1 is longer than s2 (false).	TC5	s1 = "HelloWorld", s2 = "Hello"	FALSE

❖ Boundary Test-Cases analysis:

Boundary Condition	Test Case	Input Data (Strings s1, s2)	Expected Outcome
s1 is empty, s2 is not empty (true).	TC1	s1 = "", s2 = "Hi"	TRUE
s1 is empty, s2 is empty (true).	TC2	s1 = "", s2 = ""	TRUE
s1 is equal to s2 (true).	TC3	s1 = "Test", s2 = "Test"	TRUE
s1 is a single character, s2 starts with that character (true).	TC4	s1 = "T", s2 = "Test"	TRUE
s1 is a single character, s2 does not start with that character (false).	TC5	s1 = "T", s2 = "test"	FALSE

Prefix Function :

```java
public class PrefixChecker {

    public static boolean prefix(String s1, String s2) {
        if (s1.length() > s2.length()) {
            return false;
        }
        for (int i = 0; i < s1.length(); i++) {
            if (s1.charAt(i) != s2.charAt(i)) {
                return false;
            }
        }
        return true;
    }

    public static void main(String[] args) {
        // Test cases
        System.out.println(prefix("", "Hello"));          // output true
        System.out.println(prefix("Hello", "Hello"));     // output true
        System.out.println(prefix("Hell", "Hello"));      // output true
        System.out.println(prefix("Hello", "World"));     // output false
        System.out.println(prefix("HelloWorld", "Hello"));// output false
        System.out.println(prefix("", ""));               // output true
        System.out.println(prefix("T", "Test"));          // output true
        System.out.println(prefix("T", "test"));          // output false
    }
}
```

P6: Consider again the triangle classification program (P4) with a slightly different specification: The program reads floating values from the standard input. The three values A, B, and C are interpreted as representing the lengths of the sides of a triangle. The program then prints a message to the standard output that states whether the triangle, if it can be formed, is scalene, isosceles, equilateral, or right angled. Determine the following for the above program:

a) Identify the equivalence classes for the system :

E1: All three sides are equal (equilateral triangle).
E2: Two sides are equal and one is different (isosceles triangle).
E3: All three sides are different (scalene triangle).
E4: The values cannot form a triangle due to the triangle inequality theorem (invalid).
E5: The values form a right-angled triangle.
E6: One or more sides are zero (invalid).
E7: One or more sides are negative (invalid).

b) Identify test cases to cover the identified equivalence classes. Also, explicitly mention which test case would cover which equivalence class.

Test Case	Input Values (A, B, C)	Expected Output	Covered Equivalence Class
TC1	(5.0, 5.0, 5.0)	"Equilateral"	E1
TC2	(5.0, 5.0, 3.0)	"Isosceles"	E2
TC3	(5.0, 6.0, 7.0)	"Scalene"	E3
TC4	(1.0, 2.0, 3.0)	"Invalid"	E4
TC5	(3.0, 4.0, 5.0)	"Right-angled"	E5
TC6	(0.0, 5.0, 5.0)	"Invalid"	E6
TC7	(-5.0, 5.0, 5.0)	"Invalid"	E7

c) For the boundary condition A + B > C case (scalene triangle), identify test cases to verify the boundary.

Test Case	Input Values (A, B, C)	Expected Output	Boundary Condition Description
TC1	(3.0, 4.0, 6.0)	"Scalene"	Just above the boundary
TC2	(2.0, 2.0, 3.9)	"Scalene"	Just below the boundary
TC3	(2.0, 2.0, 4.0)	"Invalid"	Boundary case

d) For the boundary condition A = C case (isosceles triangle), identify test cases to verify the boundary.

Test Case	Input Values (A, B, C)	Expected Output	Boundary Condition Description
TC1	(5.0, 3.0, 5.0)	"Isosceles"	Sides A and C are equal
TC2	(3.0, 4.0, 3.0)	"Isosceles"	Sides A and C are equal

e) For the boundary condition A = B = C case (equilateral triangle), identify test cases to verify the boundary.

Test Case	Input Values (A, B, C)	Expected Output	Boundary Condition Description
TC1	(5.0, 5.0, 5.0)	"Equilateral"	All sides equal
TC2	(1.0, 1.0, 1.0)	"Equilateral"	All sides equal

f) For the boundary condition A2 + B2 = C2 case (right-angle triangle), identify test cases to verify the boundary.

Test Case	Input Values (A, B, C)	Expected Output	Boundary Condition Description
TC1	(3.0, 4.0, 5.0)	"Right-angled"	Right angle case
TC2	(5.0, 12.0, 13.0)	"Right-angled"	Right angle case

g) For the non-triangle case, identify test cases to explore the boundary.

Test Case	Input Values (A, B, C)	Expected Output	Boundary Condition Description
TC1	(1.0, 2.0, 3.0)	"Invalid"	Cannot form a triangle
TC2	(2.0, 3.0, 5.0)	"Invalid"	Cannot form a triangle

h) For non-positive input, identify test points.

Test Case	Input Values (A, B, C)	Expected Output	Boundary Condition Description
TC1	(0.0, 0.0, 0.0)	"Invalid"	All sides are zero
TC2	(0.0, 5.0, 5.0)	"Invalid"	One side is zero
TC3	(-1.0, 2.0, 2.0)	"Invalid"	One side is negative
TC4	(3.0, -4.0, 5.0)	"Invalid"	One side is negative

Triangle Function :

```cpp
#include <iostream>
#include <cmath>
using namespace std;

const char* triangle(float A, float B, float C) {
    if (A <= 0 || B <= 0 || C <= 0 || A + B <= C || A + C <= B || B + C <= A) {
        return "Invalid";
    }
    if (fabs(pow(A, 2) + pow(B, 2) - pow(C, 2)) < 1e-6 ||
        fabs(pow(A, 2) + pow(C, 2) - pow(B, 2)) < 1e-6 ||
        fabs(pow(B, 2) + pow(C, 2) - pow(A, 2)) < 1e-6) {
        return "Right-angled";
    }

    if (A == B && B == C) {
        return "Equilateral";
    }

    if (A == B || B == C || A == C) {
        return "Isosceles";
    }
    return "Scalene";
}

int main() {
    cout << triangle(3.0, 3.0, 3.0) << endl;  // Output: Equilateral
    cout << triangle(4.0, 4.0, 5.0) << endl;  // Output: Isosceles
    cout << triangle(3.0, 4.0, 5.0) << endl;  // Output: Scalene
    cout << triangle(3.0, 4.0, 6.0) << endl;  // Output: Invalid
    cout << triangle(-1.0, 2.0, 3.0) << endl; // Output: Invalid
    cout << triangle(0.0, 2.0, 2.0) << endl;  // Output: Invalid
    cout << triangle(5.0, 12.0, 13.0) << endl; // Output: Right-angled

    return 0;
}
```